VISUALIZATION TECHNIQUES IN ATTACK GRAPHS

by

ASHOK REDDY VARIKUTI

B.S., Acharya Nagarjuna University, India, 2006

A REPORT

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2009

Approved by:

Major Professor
Xinming Ou, Ph.D.

Abstract

Attack graphs present a visual representation of all the potential vulnerabilities and attack paths in a network. They act as a vital security tool in finding the critical attack paths in the enterprise wide networks. Generated attack graphs for complex networks present thousands of attack paths to visualize and represent to the end user.

Enhancing the visualization of attack graphs by adding user interactivity will greatly improve in analyzing attack graphs and identifying the critical attack paths in the enterprise network. The layout of the attack graph can be adjusted to represent the layout of the real world enterprise network. Adding user interactivity to attack graphs is done using Prefuse, a software framework written in Java for information visualization. Prefuse is flexible and got the ability to render large amounts of data in an efficient manner.

The visualization framework for the attack graphs provides a GUI tool for interacting with attack graph. The framework is a layered architecture with two important layers, the static layer and the dynamic layer. The static layer translates the attack graph trace generated from MuLVAL into a standard graphviz dot language descriptive file. The dynamic layer translates the graphviz dot file into a graph object that can be interpreted and visualized using the prefuse software framework.

Preliminary result in this work has been published in [19].

Table of Contents

List of Figures

Acknowledgements

I would like to thank my Major Professor Dr. Xinming Ou for his constant help, encouragement and guidance throughout the project.

I would also like to thank Dr. Torben Amtoft and Dr. Gurdip Singh for serving in my committee and for their valuable cooperation during the project.

Finally, I wish to thank my family and friends for all their support and encouragement.

Dedication

This work is dedicated to my parents and sister, without whom I would not have come this far.

CHAPTER 1 - Introduction

1.1 MuLVAL

MuLVAL is an end-to-end framework and reasoning system that conducts multihost, multistage vulnerability analysis on a network [12]. It produces complete logical attack graphs along with the basic network topology information that can be further visualized. It takes as input the vulnerability information from the vulnerability database such as ICAT, network and host configuration parameters using tools such as Smart Firewall, the users of the network (Principals), the interaction model of the various components on the network and the security policy. All these information is encoded as datalog facts which are than fed into the MULVAL reasoning engine for analyzing the whole network. The MuLVAL reasoning engine scales well with the size of the network.

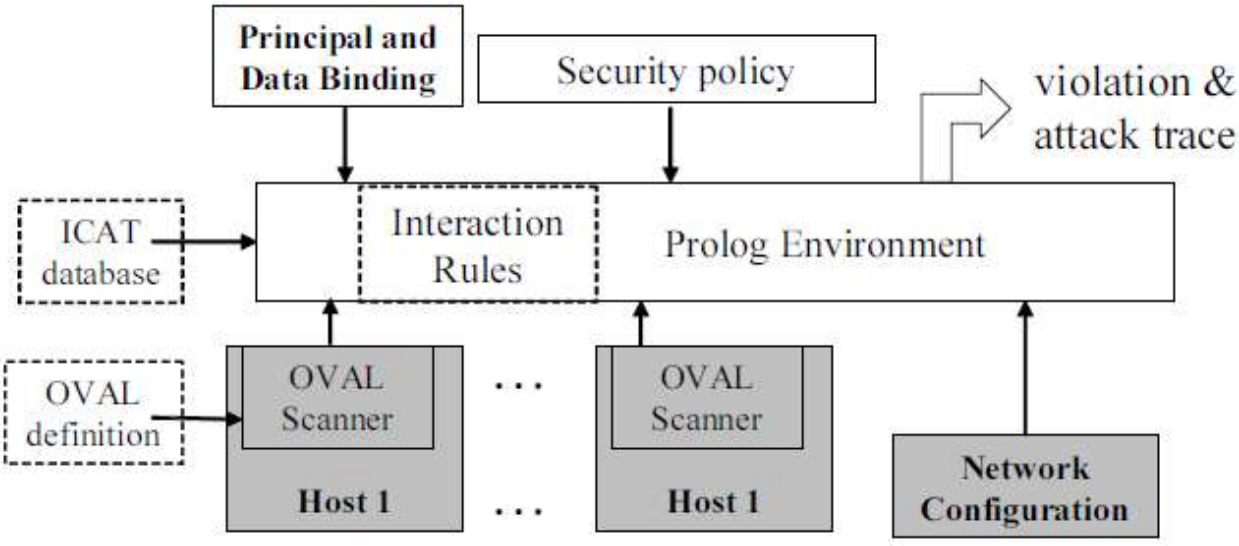

Figure 1: MuLVAL framework

1.2 Attack Graphs

Attack graphs have been used widely in various enterprise network systems to present a visual representation of all the potential vulnerabilities and attack paths [19]. They improve security by revealing the critical paths which are exploited by intruders to compromise the network and have applications in a number of areas of network security, including vulnerability analysis, intrusion alarm correlation and attack response [9][11].

Various kinds of attack graphs have been proposed in the past but most of them suffered with scalability issues and difficulty in comprehending the graph. Using model checking to compute multi-stage, multi-host attack paths in a network is the first formal method of generating attack graphs by Sheyner, et al. The state of the network is formally modeled as a collection of Boolean variables, representing configuration parameters and attacker's privileges. Attacker's actions are modeled as state-transition relations. The security property of the network is specified as a temporal formula, which can be automatically checked against the model by a model checker. Sheyner's tool outputs all the counter examples in the form of a scenario graph representing all the multi-stage, multi-host attack paths that can potentially break a network's security property. Sheyner's tool suffered with scalability issues. In an exploit dependency graph, the presence of each vulnerable component is denoted with a Boolean variable with true value. The success of an attacker is the conjunction of a set of Boolean variables (AND). Thus success of one exploit triggers another conjunction of Boolean variables to be true. Thus success of exploit is a one to one mapping from pre to post conditions.

1.2.1 MuLVAL attack graphs

MuLVAL uses a logic based approach for generating attack graphs. A node in the logical attack graph is a logical statement and the edges in the graph specify the causality relations between network configurations and an attacker's potential privileges.

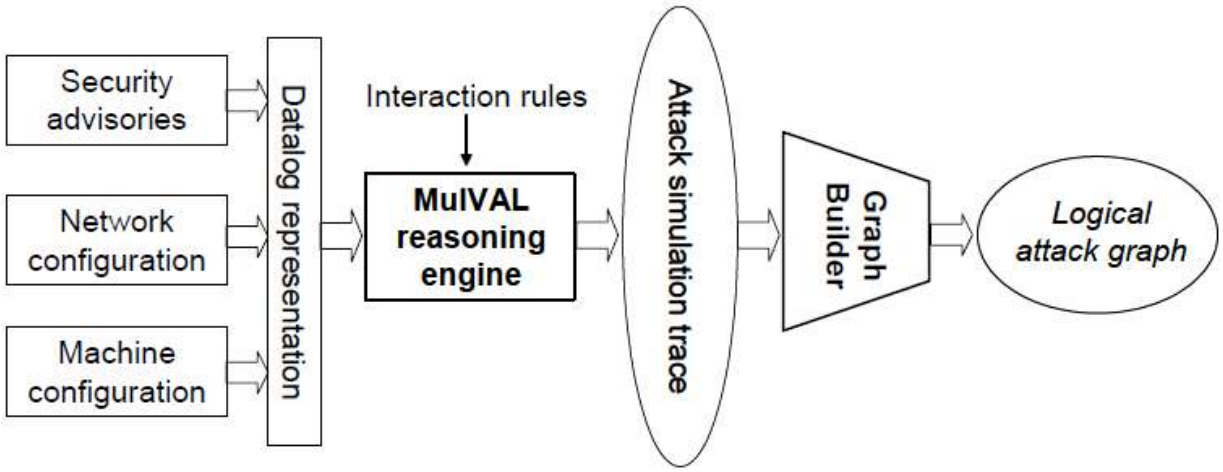

Figure 2: Logical Attack Graph Generator

Such a graph answers the question "Why an attack can happen". The sizes of these generated graphs are always polynomial to the network being analyzed and scales well with the size of the network.

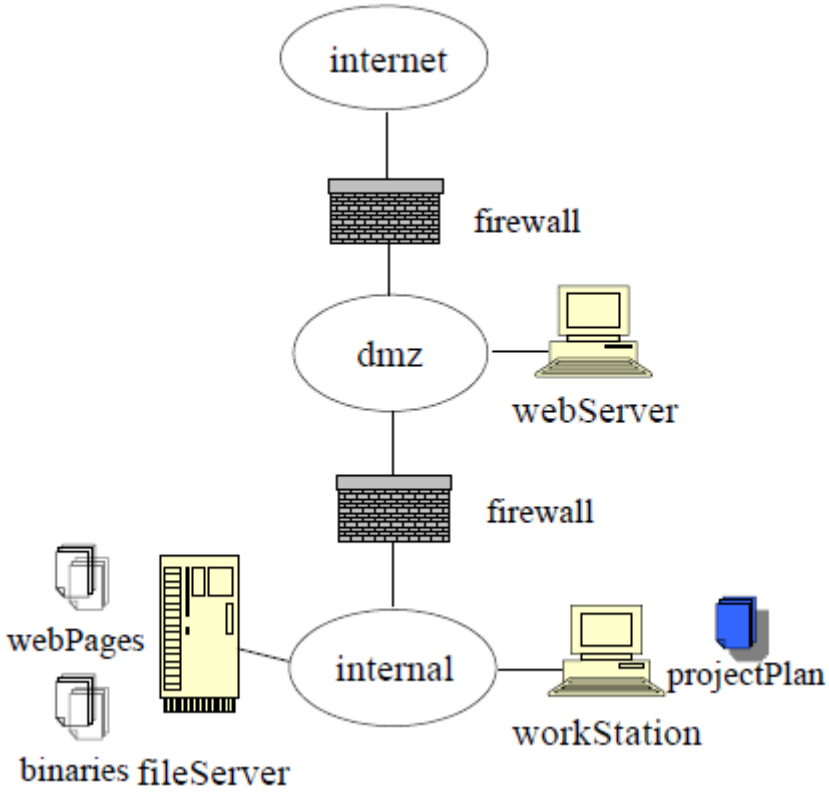

Figure 3: 3host network

For example, MuLVAL tool is run on the 3host network to find the potential attack paths. Let us assume the following attack paths are discovered by MulVAL.

An attacker first compromises webServer by remotely exploiting vulnerability CVE-2002-0392 to get local access on the server. Since webServer is allowed to access fileserver through the NFS protocol, he can then try to modify data on the file server. There are two ways to achieve this. If there are vulnerabilities in the NFS service daemons, he can try to exploit them and get local access on the machine; or if the NFS export table is not set up appropriately, he can modify files on the server through the NFS protocol by using programs like NFS Shell2. Once he can modify files on the file server, the attacker can install a Trojan-horse program in the executable binaries on fileServer that are mounted by machine workStation. The attacker can now wait for an innocent user on workStation to execute it and obtain control on the machine.

The Logical attack graph generated by MuLVAL for the above scenario is shown in Fig.5

```
<0>|--execCode(attacker,workStation,root)
   <r0>Rule5: Trojan horse installation
     <1>|--accessFile(attacker,workStation,write,/usr/local/share)
        <r1>Rule14: NFS semantics
             []-nfsMounted(workStation,/usr/local/share,fileServer,/export,read)
          <2>||--accessFile(attacker,fileServer,write,/export)
             <r2a>Rule10: execCode implies file access
                  []-fileSystemACL(fileServer,root,write,/export)
               <3>|--execCode(attacker,fileServer,root)
                  <r3>Rule3: remote exploit of a server program
                       []-networkServiceInfo(fileServer,mountd,rpc,100005,root)
                       []-vulExists(fileServer,CVE-2003-0252,mountd,
                                    remoteExploit,privEscalation)
                    <4>|--netAccess(attacker,fileServer,rpc,100005)
                       <r4>Rule6: multi-hop access
                            []-hacl(webServer,fileServer,rpc,100005)
                         <5>|--execCode(attacker,webServer,apache)
                            <r5>Rule3: remote exploit of a server program
                                 []-networkServiceInfo(webServer,httpd,tcp,80,apache)
                                 []-vulExists(webServer,CAN-2002-0392,httpd,
                                              remoteExploit,privEscalation)
                              <6>|--netAccess(attacker,webServer,tcp,80)
                                 <r6>Rule7: direct network access
                                      []-hacl(internet,webServer,tcp,80)
                                      []-located(attacker,internet)
        <r2b>Rule15: NFS shell
             []-hacl(webServer,fileServer,rpc,100003)
             []-nfsExportInfo(fileServer,/export,write,webServer)
             |--execCode(attacker,webServer,apache)==> <5>
```

Figure 4: Tree representation of 3host scenario attack graph

1.3 Problems

Even after simplifying the graphs, it's difficult to relate the layout and structure of the attack graphs to the underlying physical structure. [11]. Even for a moderate size network there can exist many attack paths which exceeds human capacity to understand and interpret the attack path information and take appropriate measures. John, *et al.* introduced techniques in reducing the complexity of attack graphs by eliminating the "Useless" attack steps that don't bring the attacker any closer to the final goal node and abstracting similar repetitive attack types into a single node with multiple sources from which only one edge will lead into the exploited node [19]. Noel, *et al.* suggested a framework in which non-overlapping attack sub graphs are aggregated to single vertices to reduce graph complexity [9].

MULVAL is an end-to-end framework and reasoning system that conducts multihost, multistage vulnerability analysis on a network [12]. The attack graph data when visualized using a simple graphviz dot language produces a static image which is too complex to understand and interpret to the human user. After applying the trimming algorithm to trim a large portion of the attack graph [19] generated by MULVAL [12] that is not needed for the human user to understand the main security problem, the complexity of the attack graph for a moderately large network is still overwhelming for a human user. The layout of the graph is completely unrelated to the underlying topology of the network. Large number of edges, nodes and subnets in the network results in overlapping nodes and edges increasing the complexity and resulting in a graph that is beyond the comprehension of the human user. The user besides can't interact with the graph.

1.4 Contribution

Mapping attack graph data into the underlying network topology and formalizing a visualization framework for providing user interactivity for better understanding and analysis of logical attack graphs. Embedding interaction to the graph can realize their full potential. It gives the human user the chance of viewing the attack graph in multiple dimensions and interpret the attack path and security vulnerabilities in a more precise and convenient manner and take precautionary steps accordingly. Besides interactivity, representing the actual node and subnet data with images and highlighting the attack steps reduces the gap between the actual network

and the visual representation of the network making them look alike seamlessly and allows the user to concentrate on the most critical data set in the graph. This is an extension of previous work which is published in [19]. Work in [19] explores various ways of improving the visualization in attack graphs by trimming attack paths and visualizing the attack graph using graphviz. But even after trimming useless attack steps, for a large network with thousands of nodes the network gets complex. This report primarily addresses in reducing the complexity in attack graphs using visualization techniques.

CHAPTER 2 - Survey of network visualization toolkits

Many toolkits exist in the academia community for visualizing large and complex network graphs. The advantage of using an existing toolkit instead of designing from the scratch is that these are widely used and accepted in the community for many years and are proven to work efficiently, well organized and have classes that can be extended and customized for specific applications. The following are the visualization tools that I found to be very useful for visualizing attack graphs which can be extended to meet attack graphs specific requirements.

2.1 JUNG - java Universal Network/Graph Framework

JUNG Framework is an open-source Java-based software library that has been developed specifically as a common and extendible language for the manipulation, analysis, and visualization of data that can be represented as a graph or network [20]. JUNG is a good platform for exploratory data analysis on relational data sets. The salient futures of JUNG are:

- Can be used to represent various types of graphs such as directed and undirected graphs, multi-modal graphs (graphs which contain more than one type of vertex or edge), graphs with parallel edges (also known as multigraphs), and hypergraphs (which contain hyperedges, each of which may connect any number of vertices).
- Classes implementing number of algorithms from graph theory, exploratory data analysis, social network analysis, and machine learning. These include routines for clustering, decomposition, optimization, random graph generation, statistical analysis, and calculation of network distances, flows, and ranking measures.
- A visualization framework that makes it easy to construct tools for the interactive exploration of network data. Users can choose among the provided layout and rendering algorithms, or use the framework to create their own custom algorithms.
- Mechanisms for annotating graphs, entities, and relations with metadata. These capabilities facilitate the creation of analytic tools for complex data sets that can examine the relations between entities, as well as the metadata attached to each entity and relation.

2.2 Piccolo

Piccolo is a toolkit that supports the development of 2D structured graphics programs, in general, and zoomable user interfaces, in particular. The toolkit maintains a hierarchal structure of objects and cameras, which allows the application developer to orient, group and manipulate objects in meaningful ways [21].

The Piccolo visualization toolkit produces effective visualizations of data using tree and fisheye layouts which are easily adapted to user data. The tree layout gives users access to large amounts of hierarchal data, but presents it in a format which allows them to focus on particular areas of interest. Fisheye layout gives user visibility to a large amount of data at a high level using a grid pattern. When users select an area of interest the layout reveals more detail.

The Piccolo toolkit includes numerous beneficial functions that are helpful for visualizations. Piccolo also provides the necessary design and structure to support a network visualization framework. These features, along with a structured approach to adding custom animations, make Piccolo a good toolkit to use with the network visualization framework. The only drawback of Piccolo is the monolithic design approach. This approach creates large top-level classes. Expanding the framework then requires sifting through lines and lines of code to add functionality to the major top-level classes of the toolkit. Because it is anticipated that this framework will be expanded through future research the monolithic Piccolo toolkit is less desirable.

2.3 Graphviz

Graphviz is a toolkit of libraries and programs for creating, filtering, displaying and interacting with graphs. The toolkit contains base libraries for handling attributed graphs; a collection of graph layout algorithms; platform-dependent frontends; and a complement of file-stream graph processors [18].

Libraries		
	Libgraph	attributed graph data structures and I/O
	Dynagraph	incremental layout library
Layout Tools		
	Dot	hierarchical layout
	Neato	"symmetric" layout
Graphical Tools		
	Dotty	programmable interactive graph editor
	Tcldot, Webdot	related interactive front ends
	Grappa	compatible graph package in Java
	Montage	generic ActiveX diagram container
	Tcldg/Dged	Tcl/Tk graph editor for incremental layout
Graph Filters		
	Gpr	generic graph filter
	Sccmap	decomposes graphs into strongly-connected components
	Colorize	computes node colors from initial seeds
	Unflatten	adjusts edges to improve aspect ratio of hierarchical layouts

Figure 5: Graphviz components

Dot (for directed graphs) and neato, fdp(for undirected graphs) are the two main graphviz layout tools. Dot generates hierarchical drawings of directed graphs with edges aimed in same direction (top to bottom, or left to right) and also avoids edge crossings. Neato and fdp make ``spring model" layouts. Neato uses the Kamada-Kawai algorithm for layout whereas fdp implements the Fruchterman-Reingold heuristic including a multigrid solver that handles larger graphs and clustered undirected graphs. Twopi generates "radial layouts". The nodes are placed on concentric circles depending on their distance from a given root node. Circo generates circular layout which is suitable for certain diagrams of multiple cyclic structures such as certain telecommunications networks.

2.4 Prefuse

Prefuse is an extensible software toolkit for helping software developers create interactive information visualization applications using the Java programming language. Prefuse simplies the processes of data handling, representation, and mapping to on-screen displays as

well as crafting direct manipulation interactions with the visualization. The prefuse visualization toolkit is well designed and supported. SourceForge.net hosts online support through a user forum with a large number of prefuse developers including the toolkit's author Jefrey Heer. The prefuse toolkit follows the model-view controller design pattern. The prefuse action, render, and display packages constitute the view and prefuse control and data.query packages make-up the controller. Figure 2.12 shows a high level view of the prefuse toolkit and highlights the model-view controller design pattern elements. Prefuse organizes data structures into two main hierarchies: the Tuple and TupleSet interfaces.

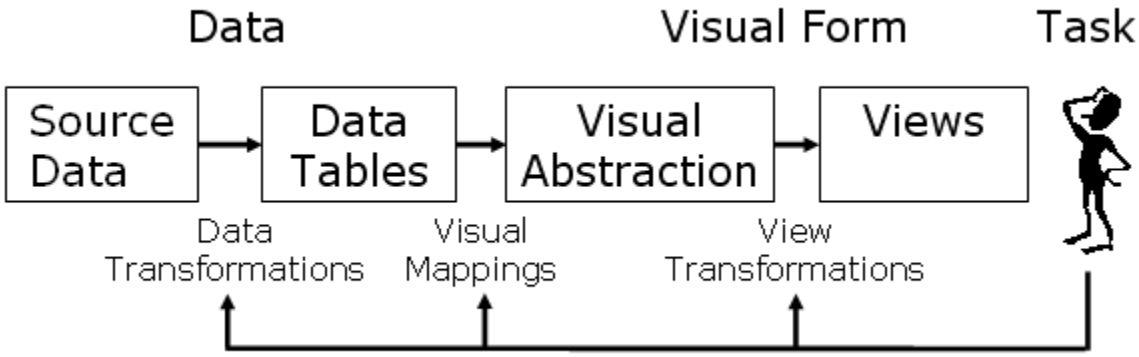

Figure 6: Prefuse reference model

From these two interfaces branch all the concrete data structures and collections of data structures that form the prefuse model. Prefuse provides a rich polylithic class hierarchy of data structures for visualizations. Prefuse data structures include the commonly used structures graphs, tables, edges and nodes. VisualGraph, VisualTable, and VisualItem are superclasses of these data structures and contain important display information. VisualGraph denotes the topology of the network (nodes and edges) and VisualTable holds a collection of VisualItems. Modifying the Visual objects creates a corresponding change in how the prefuse visualization toolkit renders each object on screen.

CHAPTER 3 - Formalization of the visualization problem

3.1 Importance of formal specification

Various visualization tools exist in the visualization community each having its own advantages and dis-advantages. A formal specification for the framework can be exploited in implementing different parts of the framework with various visualization tools thus extracting the highlight features specific to each respective tool. This makes the framework tool independent and can open a wide array of opportunities from using multiple toolkits to creating a customized visualization toolkit. For example, the salient feature of Piccolo toolkit is using fisheye layout and tree layouts and prefuse has zooming, panning and dragging capabilities as one of its salient features. Two different implementations can be created using the same formal specification using Piccolo and prefuse respectively depending upon the desired features for a particular topology or the human user.

3.2 Overview of the visualization architecture

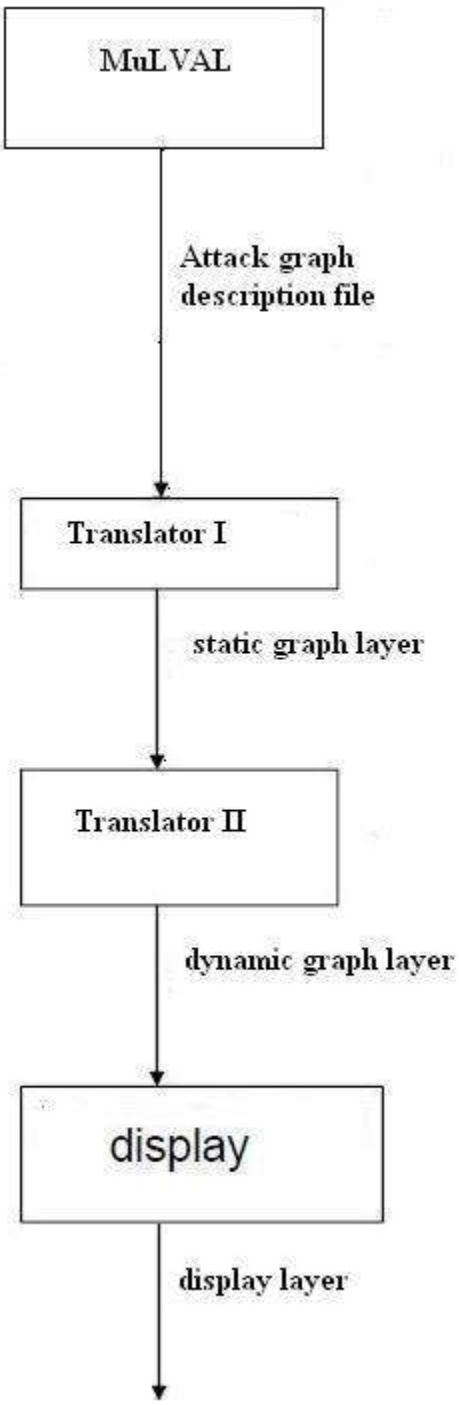

Figure 7: Visualization architecture

The architecture provides a comprehensive framework for visualizing attack graphs. The trace generated from MulVAL is very difficult to understand and visualize. Mapping the

underlying network topology from the MulVAL trace and seamlessly giving complete control of the network layout to the user is the main objective. Graphviz is an open source graph visualization tool written and developed by AT&T labs. Initially MulVAL generates the trace of a logical attack graph when given various parameters as input such as network configuration parameters, security policies etc. This trace is further subjected to trimming techniques as described by Homer et al. The refined trace consists only of the network layout information and all the possible attack paths between various nodes and subnets in the network. It can be considered as a high level abstraction of the logical attack graph trace wherein only essential details such as attack paths and topology information is represented which helps a naïve user such as an administrator to easily identify the vulnerabilities in the network and take suitable measures accordingly.

One of the great disadvantages with the layout generated from graphviz is that the layout can't be changed and moreover it produces plain static images which don't provide any user interactive capabilities to analyze the graph. But the initial layout of the graph generated by graphviz using FDP provides a standard platform for displaying attack graphs in which overlappings are minimized. This initial layout information when combined with interactive capabilities will produce high quality graphs which can provide the user full control over them. Hence the need for dynamic graph layer in the architecture. This translation is for mapping the graphviz layout information (dot format) to a prefuse graph data structure format.

The initial layout of the prefuse graph corresponds to the exact co-ordinates of each of the nodes and edges as provided by graphviz. The following command produces the layout information along with the exact screen (x,y) co-ordinates for each of the graph structure node, edge and subgraph.

This generated file is fed into the translator which emits a corresponding prefuse graph with the layout information taken from the generated file.

CHAPTER 4 - Data Format

4.1 Static graph layer

The static graph layer translates the attack graph description file.into graphviz dot file. The format of the input data for the layer with an enterprise network taken as an example can be defined as follows:-

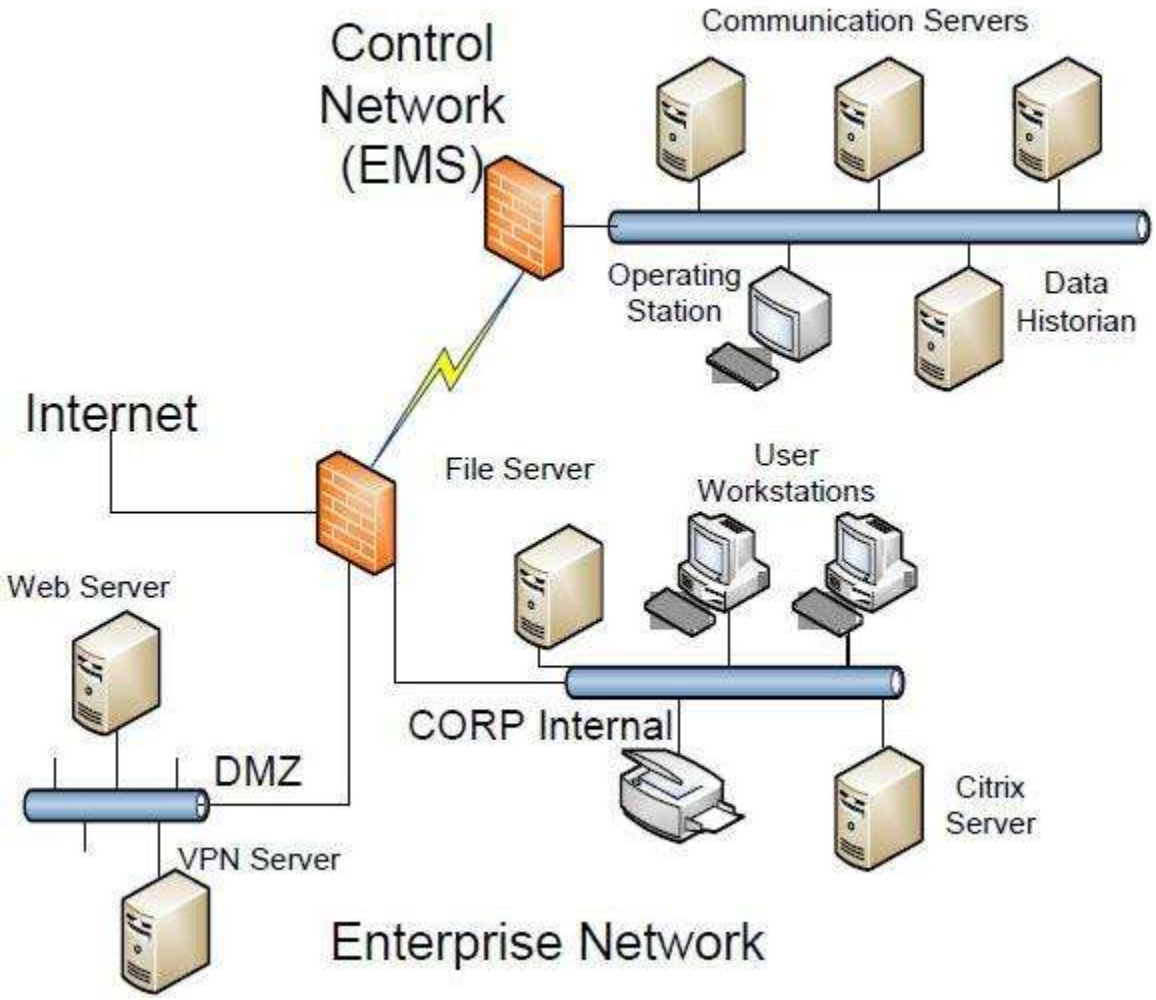

Figure 8: Enterprise Network

Node (V):- The node is the most basic data type. It maps to an actual node in the real world enterprise network. <citrixServer>, <dataHistorian>,<webServer> etc.. are the nodes in the EN.

Gateway (G):- A gateway G typically maps to a firewall in a real world enterprise network.

Subnet(S):- A subnet S is defined as follows:-

$\forall$ s:Subnet | $\exists$ (V1,V2..Vn) $\in$ Node | s $\in$ (V1,V2…..Vn).

A subnet s is just a grouping of different nodes in the network. All the nodes in the subnet implicitly mean that all of them are interconnected seamlessly. Eg:- DMZ, CORP Internal and Control Network are the 3 subnets in the network.

Connection (C):- A connection C is defined as the link between <subnet, gateway> $\forall$ c:Connection | $\exists$ (S1,S2..Sn) $\in$ subNet, (G1,G2..Gn) $\in$ Gateway | c $\in$ (<S1, G1>, <S2, G2>,….<Sn,Gn>). Eg:- {(<DMZ>,<firewall>),(<CORP Internal>,<firewall>)(<Control Network>,<firewall>) are the three connections in the network.

AttackStep (AS):- An attack step AS is defined as the link between <node,node>

$\forall$ AS:AttackStep | $\exists$ (N1,N2..Nn) $\in$ Node | AS $\in$ (<N1, N2>, <N2, N3>,….<Nn-1,Nn>)

An attack step represents a critical attack path between 2 different nodes in the network. This is the most important information which is the direct consequence of MuLVAL generated attack step trace. Eg:- (<citrixServer,dataHistorian>) can represent a possible attack step in the network.

4.2 Dynamic graph layer

The dynamic graph layer translates the graphviz dot file into a graph interpretable by prefuse framework. The format of the input data for the layer can be defined as follows:-

Graph (G):- The directed graph G<N, E> with set of nodes ‘N’ and set of edges ‘E’.

Node (V):- A node **V** of the graph G is defined as a single tuple t’ in the data table. The columns in the node table denote the node attributes.

A(N):- set of attributes for the Node N.

$\forall$ n(t):Node | A(n) = {id, name, insubnet, label} ;

id:- returns the unique id generated for this node

name(id):- This function returns the name assigned to the node with ID "id".

Insubnet(id):- This is a boolean function which returns true when the node with ID "id" is part of a subnet else returns false. In our network topology, the incoming edges to the nodes work according to the following psedo algorithm:-

If(insubnet(id) == true)

    Truncate the incoming edges to this node at the boundary box of the subnet in which this node in enclosed within;

Else

    Truncate the incoming edges to this node at the boundary box in which the node is enclosed within.

Label(id):- This function returns the label assigned to this node. The label can consist any meaningful name. The range of characters allowed in the label is:-

Edge E (V, V'):- An edge **E (V, V')** in the graph G is defined as a single tuple t' in the datatable Dt. The columns in the table denote the edge attributes.

E(N):- set of attributes for the Edge E

$\forall$ e(t):Edge | E(n) = {id, source, destination, label};

edge(id):- returns the unique id generated for this edge

range(edge(id)) = {1,2….N} i.e., it accepts values from domain of natual numbers

Source(this.id)):- returns the id of the node that the edge is incident from.

Destination(this.id):- returns the id of the node that the edge is incident to.

Label(this.id):- returns the label assigned to this edge.

ViewTable(Vt):- Vt is defined as a set of tuples t' where visual properties are columns containing visual properties such as location, bounding box, colors, size, and font.

A(Vt):- set of attributes for the Viewtable Vt.

$\forall$ v(t):ViewTable | v(t) = {id, bounding, color, size,font,shape,group};

group(id) = {node, edge, aggregate,graph};
semantics:- the group function that returns the visualization group the id belongs to. The id can belong to a node, an edge or an aggregate.

bounding(id):- This returns the (xbl,ybl,xbr,ybr,xbt,ybt,xtr,ytr) (four (x,y) co-ordinates at the four corners) co-ordinates for the rectangular bounding box enclosing the particular node or aggregate. For an edge, this returns (xs,ys,xs+2,ys+2,xs+4,ys+4…xs+n,ys+n) co-ordinates along the edge that dictates the shape and curve for the edge. This dictates the geometric shapes for node, edges or rectangles.
Shape(id) = {rectangle, square, polygon}
color(id) = {RGB comination}

ViewNode(VI):- An instance of a ViewNode VI' with the tuple instance t' in the visual table Vt where Vt(group(id)) = node and and id gives the viewnode id.

∀ v(t):ViewNode | v(t) = { id, bounding(id), color(id), size(id),font(id),shape(id),group(id)="node"};

ViewEdge(EI):- An instance of a ViewEdge EI' with the tuple instance t' in the visual table Vt where Vt(group(id)) = edge and and id gives the viewedge id.

∀ v(t):ViewEdge | v(t) = { id, bounding(id), color(id), size(id), font(id), shape(id), group(id)="edge"};

ViewSubnetTable St: - At is an instance of visualtable Vt where each tuple is defined as t''<v1, v2….visualproperties> representing the VisualItem instance Vi contained in the visualtable Vt. This table also contains the mapping Vi → AI denoting the belongs to relationship for each visualitem to an aggregateitem instance.

Visualization Vg: - Vg is an abstract data structure (table, graph or tree) which contains the mappings Node V → Vt, Edge E → Et and aggregate A → At.

CHAPTER 5 - Translation and Display

5.1 Display Layer

In this layer the generated prefuse graph object is visualized on the screen.

Pseudo algorithm for display:-

Algorithm 5.1.1 Pseudocode for displaying the graph and user interactivity

```
1: Initialize Graph G
2: for all Nodes N ∈ G do
3:     Add renderers (label, size, text, shape, color) to N
4:     Load images in N
5: end for
6: for all Edges E ∈ G do
7:     Add renderers (shape, label) to E
8: end for
9: for all Aggregates A ∈ G do
10:    add renderer shape to A
11:    add drag control to A
12: end for
13: Add zoom control, pan control and control listeners, layouts to G
```

Figure 9: Pseudocode for displaying the graph and user interactivity

Figure 9 lists the skeleton steps involved in adding user interactivity to the graph. Initially renderer's for rendering visual properties for edges, nodes and aggregates such as color, shape, size, font, bounding box are added which dictates the visual properties of the graph. Each aggregate additionally has a drag control to it which gives control of layout of the network for the user. The zoom control helps in zooming into the network display and helps the user in comprehending the attack information when network scales to 1000's of nodes and graphs. The pan control helps in moving the network display in horizontal directions (sideways). Various controls listeners dictate the actions for various mouse, keyboard events such as on clicking a node, clicking a edge, on entering an aggregate etc.

5.2 Dynamic Layer Translation

The translation in this layer mainly consists of generating a prefuse software framework compatible graph object.

Algorithm 5.2.1 Psuedocode for constructing layout of the graph

```
1: Initialize empty graph G
2: dotFile ← File denoting the graph layout information in dot format
3: for all tokens t ∈ dotFile do
4:       Create an abstract syntax tree T from t
5: End for
6: Construct graph G from AST T
7: Add G in Vg
8: for all edges e ∈ G do
9:       Add edge decorator to e
10: End for
11: Do the layout action for the graph G
```

Figure 10: Psuedocode for constructing layout of the graph

Figure 10 creates the graph object G from abstract syntax tree AT. AT is generated by feeding the tokens extracted from the contentes of the dotFile to the translator. We than recursively walk the AT to perform actions corresponding to each token input sequence to generate prefuse graph G. Adding graph to the visualization object Vg automatically adds the graph to the visualization group. Edge decorators denote the color, shape and text style of each label assigned to the edges. Layout action performs the actual layout of graph.

5.3 Static Layer Translation

Grammar Rules

Algorithm 5.3.1 Pseudo-code for graph initialization

```
1: Gloval variables: - isClusterID, nodeTable, edgeTable
2:      //Pseudo algorithm 1 for adding graph
3:      Input ← M
4:      Output ← Gd, Gg
5:      Ts ← {ID, Edge, SubgraphID, "DIGRAPH", "GRAPH"}
6:       Gv ← Visualization object                    Nr ← Node function return
7:      M ← Current matching pointer                 Gg ← Visualization graph group
8:      Vt ← ViewTable                               Gn ← Visualization node group
9:      Ga ← Visualization aggregate group
10:     At ← Aggregate Table                         Er ← edge function return
11:     G ← {Nt, Et} {Initialize the graph}
12:     If M n Ts ∈ {"DIGRAPH"} then
13:     Gd ← {Vt} {initialize a digraph with viewtables}
14:             Gg ← Gv.add(Gd)
15:     Else
16:     Gu ← {Vt} {initialize an undirected graph with viewtables}
17:             Gg ← Gv.add(Gu)
18:     End if
```

Figure 11: Pseudo-code for graph initialization

Explanation: - Figure 11 creates an empty graph and initializes it with nodes and edges from the viewtables. Gv is the graph visualization object. Addition of the newly created graph to the visualization object creates an interactive version of the graph that can be zoomed, panned and subjected to various operations.

Algorithm 5.3.2 Pseudo-code algorithm for a matching ID (i.e., nodes)

```
1:    Input parameters: - Graph Object Go, SubnetID subID
2:    Return parameter: - Nr
3:    If M n Ts ∈ {ID} Then
4:        If ∃ node | node.ID = ID then {Just return the matched nodeID}
5:        Nr ← Nr U ID
6:    Else
7:    If ∃ aggregateItem | aggregateItem.ID = ID then {Return the aggregateItem ID} than
8:        Nr ← Nr U ID
9:        isClusterID ← true
10:   Else
11:           If Go ∈ At than
12:               Nr = Nt.addrow()
13:               Setdefaultnodeattributes(ID,Nt)
14:               Nt.set("nodeID", ID)
15:               Nt.set("subnetNode", true)
16:               Nt.set("subnetID", subID)
17:           Else
18:               Nr = Nt.addrow()
19:               Setdefaultnodeattributes(ID,Nt)
20:               Nt.set("nodeID", ID)
21:               Nt.set("subnetNode", false)
22:           End if
23:       End if
24:   End if
```

Figure 12: Pseudo-code algorithm for a matching ID

Algorithm 5.3.3 Pseudo-code algorithm for matching edges

```
1:    Globalvariable: - isClusterID
2:    Input parameters: - Graph Object Go, SubnetID subID
3:    Local variables: - Eid, Esource, Edest
4:    Return parameter: - Nr
5:        Esource ← call algorithm 3 recursively or algorithm 2 to return the source nodeID
6:        Edest ← call algorithm 2 to return the destination node ID
7:        Eid = Et.addrow()
8:        Setdefaultedgeattributes(Eid,Et)
9:        Et.set(Eid, "source", Esource)
10:       Et.set(Eid," destination", Edest)
11:           If (isClusterID)
12:               Et.set(Eid,"isaggregateEdge", true)
13:               isClusterID ← false
14:           End IF
15:       Nr ← Edest
```

Figure 13: Pseudo-code algorithm for matching edges

Algorithm 5.3.4 Psuedo-algorithm for matching ID of aggregate items

```
1:    Input Parameters: - Graph Object Go
2:    Local variables: - Aitem ← aggregate item
3:    If M n Ts ∈ {SubgraphID} Than
4:        Aitem ← At.addItem()
5:        Aitem.set("id",SubgraphID)
6:        Nset ← Gg.nodes()
7:    For each Nset | Nset != { } do
8:        If Node ∈ Nset | Node.ID ∈SubgraphID than
9:            Aitem ← Aitem U Node
10:   End For each
```

Figure 14: Psuedo-algorithm for matching ID of aggregate items

CHAPTER 6 - Implementation and testing

6.1 Project Build Model

The whole visualization architecture is implemented as a java project. The project is build and managed using apache maven. The maven plugin is downloaded and installed in eclipse IDE. The project structure of the whole model is given below.

DOTViewer

- target/generated-sources/antlr
    - Prefuse.parser.dotparser (package to store the compiled Antlr grammar files)
- src/main/java
    - The whole prefuse package
    - prefuse.parser.dotparser (package containing the main GUI interface)
    - prefuse.parser.dotparser.util (custom packages for renderer's etc..)
- src/main/antlr
    - Input antlr grammar files
- src/main/resources
    - images and input sample graphviz dot files
- pom.xml

The src/main/java and src/main/antlr directories contain all the project source code. The POM (project object model) contains all necessary information about the project, as well as configurations of plugins to be used during the build process.

6.2 Core prefuse classes

Visual Mappings

The visual mappings from the source data to onscreen VisualItems is achieved using the Visualization class.

Visualization *(my_Vis)*: - new Visualization ( )

Visualization: - This class takes the datasets of type aggregates, graph and maps into a corresponding instance of VisualTables, VisualGraphs. Addition of edges and nodes to this class creates an interactive, visual realization of the node and edge tuples respectively. In the DOTTree class, a visualization object is created and the graph is added using the graph constructor. The nodetable and edgetable are declared as static. Any modification to the node and edge tables changes the visual representation of the whole graph.

Operations (*my_Vis*): -

- items(String group, Predicate filter): - Returns an iterator over all items in the given group which match the given Predicate filter. The group can be a string identifying edges, nodes or aggregates in the graph.

Visual Groupings

Various groupings can be created to perform operations on specific parts of the graph.

- DOTTree.GRAPHGRP: - String used to denote the entire graph. Specifically used to filter and identify particular sections of the graph using expression parsing techniques.
- DOTTree.NODESGRP: - String used to denote the nodes in the graph. Can be used to perform specific operations of interest on the nodes of the graph. For example,

```
Iterator nodeIter = (new Visualization()).items(DOTTree.NODESGRP,
ExpressionParser.predicate("[nodeID] = '" + “dataHistorian” + "'"));
```

Here, the items method filters out the node tuples from the graph with ID “dataHistorian”. It returns an iterator over the node tuples.

- DOTTree.EDGESGRP: - String used to denote the edges in the graph. Can be used to perform specific operations of interest on the edges of the graph.
- DOTTree.AGGRGRP: - String used to denote the aggregates (subnets) in the graph. Can be used to perform specific operations of interest on the aggregates (subnets) of the graph.

CHAPTER 7 - Conclusion and future work

The attack graph visualization framework improves interactivity with attack graphs. It gives the power even for a naive network administrator to analyze attack graphs and identify the critical attack paths in the network. Networks scaling to thousands of nodes can be easily analyzed using panning and zooming features in prefuse package. The package can be easily extended over time to add extra features.

The default layout algorithms provided by prefuse package doesn't resolve the problem of mapping the original network layout to the corresponding visualization of attack graph on the user screen. They generate an unstable layout and makes interaction almost impossible. The static layer in the visualization framework generates a stable layout for the attack graph.

Large number of toolkits exist for visualizing network graphs. A universal framework that can integrate the salient features of each toolkit into a single visualization framework can give various dimensions of user interactivity to attack graphs. The proposed architecture in this paper is limited to interactive features existing in prefuse visualization toolkit.

References

1. Laura P. Swiler, Cynthia Phillips, David Ellis, and Stefan Chakerian. Computer-attack graph generation tool. In *DARPA Information Survivability Conference and Exposition (DISCEX II'01)*, volume 2, June 2001.

2. Oleg Sheyner, Joshua Haines, Somesh J{Belue, 2008 #8}ha, Richard Lippmann, and Jeannette M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 254–265, 2002.

3. Paul Ammann, DumindaWijesekera, and Saket Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of 9th ACM Conference on Computer and Communications Security*, Washington, DC, November 2002.

4. Sushil Jajodia, Steven Noel, and Brian O'Berry. Topological analysis of network attack vulnerability. In V. Kumar, J. Srivastava, and A. Lazarevic, editors, *Managing Cyber Threats: Issues, Approaches and Challanges*, chapter 5. Kluwer Academic Publisher, 2003.

5. Richard Lippmann and Kyle W. Ingols. An annotated review of past papers on attack graphs. Technical report, MIT Lincoln Laboratory, March 2005.

6. Kyle Ingols, Richard Lippmann, and Keith Piwowarski. Practical attack graph generation for network defense. In *22nd Annual Computer Security Applications Conference (ACSAC)*, Miami Beach, Florida, December 2006.

7. Xinming Ou, Wayne F. Boyer, and Miles A. McQueen. A scalable approach to attack graph generation. In *13th ACM Conference on Computer and Communications Security (CCS)*, pages 336–345, 2006.

8. Wei Li, Rayford B. Vaughn, and Yoginder S. Dandass. An approach to model network exploitations using exploitation graphs. *SIMULATION*, 82(8):523–541, 2006.

9. Steven Noel and Sushil Jajodia. Managing attack graph complexity through visual hierarchical aggregation. In *VizSEC/DMSEC'04: Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, pages 109–118, New York, NY, USA, 2004. ACM Press.

10. Steven Noel, Michael Jacobs, Pramod Kalapa, and Sushil Jajodia. Multiple coordinated views for network attack graphs. In *IEEE Workshop on Visualization for Computer Security (VizSEC 2005)*, 2005.

11. Richard Lippmann, Leevar Williams, and Kyle Ingols. An interactive attack graph cascade and reachability display. In *IEEE Workshop on Visualization for Computer Security (VizSEC 2007)*, 2007.

12. Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. MulVAL: A logic-based network security analyzer. In *14th USENIX Security Symposium*, 2005.

13. Somesh Jha, Oleg Sheyner, and Jeannette M. Wing. Two formal analyses of attack graphs. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, pages 49–63, Nova Scotia, Canada, June 2002.

14. Richard P. Lippmann, KyleW. Ingols, Chris Scott, Keith Piwowarski, Kendra Kratkiewicz,Michael Artz, and Robert Cunningham. Evaluating and strengthening enterprise network security using attack graphs. Technical Report ESC-TR-2005-064, MIT Lincoln Laboratory, October 2005.

15. Richard Lippmann, Kyle Ingols, Chris Scott, Keith Piwowarski, Kendra Kratkiewicz,Mike Artz, and Robert Cunningham. Validating and restoring defense in depth using attack graphs. In *Military Communications Conference (MILCOM)*, Washington, DC, U.S.A., October 2006.

16. Vaibhav Mehta, Constantinos Bartzis, Haifeng Zhu, Edmund Clarke, and Jeannette Wing. Ranking attack graphs. In *Proceedings of Recent Advances in Intrusion Detection (RAID)*, September 2006.

17. Lingyu Wang, Anoop Singhal, and Sushil Jajodia. Measuring network security using attack graphs. In *Third Workshop on Quality of Protection (QoP)*, 2007.

18. Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. In *Software - Practice and Experience*., **00**(S1), 1–5 (1999).

19. John Homer, Ashok Varikuti, Xinming Ou, and Miles A. McQueen. Improving attack graph visualization through data reduction and attack grouping**.** In *5th International Workshop on Visualization for Cyber Security (VizSEC 2008),* Cambridge, MA, U.S.A., September 2008.

20. Joshua O'Madadhain, Danyel Fisher, Padhraic Smyth, Padhraic Smyth and Yan-Biao Boey. Analysis and Visualization of Network Data using JUNG. In *Journal of Statistical Software.*

21. 'Human Computer Interaction Lab' – "Piccolo Toolkit". URL http://www.cs.umd.edu/hcil/jazz/. February 5, 2008.